Das Portable Forth Environment

Eine einfache Realisierung von Forth in C

Dirk Zoller

Mannheim, 14. April 1994

Zusammenfassung

Es wird eine Implementation des draft proposed American National Standard (dpANS) für die Sprache Forth beschrieben. Das beschriebene System hat den Arbeitsnamen **pfe**, setzt den dpANS vollständig um und ist selbst in ANSI-C geschrieben, so daß es leicht portabel ist auf verschiedene Rechnerumgebungen. Obwohl das System den aktuellen Standard realisiert, ist der Ansatz in vieler Hinsicht traditionell und so einfach wie möglich. Dieses Papier erläutert einige der Designentscheidungen, die dabei zu treffen waren, und gibt Hinweise zum Einsatz und zur Portierung von **pfe**.[1]

1 Ziele

Der kommende ANSI-Standard [1] für die Sprache Forth ist wesentlich umfangreicher als alle bisherigen Standards für diese Sprache. Gleichzeitig vermeidet er die Regelung von Implementationsdetails, eine Eigenschaft früherer Standards, die zunehmend deren Akzeptanz beeinträchtigte.

Implementatoren, die bisher aus dem engen Korsett des fig-Modells oder dem 16-Bit-Diktat des FORTH-83-Standards ausbrachen, sahen sich auf sich selbst gestellt und waren gezwungen, nicht nur in Implementationsdetails abzuweichen, sondern wesentliche Spracheigenschaften frei zu gestalten. Die Folge war eine Phase des Wildwuchses, wo kein Forth wie das andere war. Sie geht nun hoffentlich zu Ende.

Verfügbarkeit und Portabilität Jetzt ist es wichtig, daß treue und vollständige Implementationen des Standards bereitstehen, die es ermöglichen, Quelltexte an den Standard anzupassen, und die auf die noch zögernden Anbieter den notwendigen Anpassungsdruck ausüben.

Der Umfang des Standards macht seine vollständige Umsetzung mit den bisherigen Methoden – in Assembler oder mittels eines Metacompilers – zu einer nichttrivialen Aufgabe, für die die Anbieter offensichtlich noch Zeit brauchen.

pfe wurde dagegen aus zwei Gründen in ANSI-C geschrieben: Es sollte in vollem Umfang rasch zur Verfügung stehen und es sollte auf sovielen Plattformen wie möglich – und das heißt zuallererst auf UNIX – zur Verfügung stehen.

pfe läuft z.Z. auf verschiedenen UNIX-Derivaten[2] sowie auf DOS (ab 386) und OS/2 ab 2.0. Anpassungen an weitere 32-Bit-Systeme, deren C-Compiler gewisse UNIX-ähliche Funktionen in der Library haben,[3] dürften einfach sein. Seit fig-Forth ist kein freies Forth-System mehr so gleichartig auf so vielen verschiedenen Rechnern gelaufen.

Vertrautheit Damit ein zur Verfügung stehendes System aber auch benutzt wird, müssen sich potentielle Benutzer darin wohlfühlen. In **pfe** wurde deshalb einiger Aufwand getrieben, den interaktiven "Look and feel" von Forth-Systemen auf typischen Einplatzsystemen (PC, Atari,...) trotz der Portabilität zu

[1] **pfe** kann per anonymous ftp bezogen werden: roxi.rz.fht-mannheim.de:/pub/languages/forth/pfe-x.x.x.tar.gz

[2] bisher sind erprobt: Linux, AIX 3.2x, HP-UX 8.x, Domain-OS, Sun-OS, Ultrix 4.3 RISC, Irix.

[3] In erster Näherung kann man vermuten, daß **pfe** mit geringem Aufwand portabel ist auf ein System, wenn dort auch GNU-C läuft.

erhalten. So wird eine Tastatureingabe nicht einfach mit der C-Funktion *gets()* abgeholt, sondern das Verhalten von `EXPECT` nachgebildet. Die Kommandozeile erlaubt das Zurückholen und das Edieren alter Eingaben. Cursor-Positionierung und Text-Attribute sind möglich, Funktionstasten können mit Forth-Worten belegt werden und ein vollständiger Block-Editor ist integriert.

Zum Wohlfühlen in einem System gehört auch, daß es vertraut ist. Deshalb wurden bekannte Worte, die nicht den Weg in den Standard gefunden haben, trotzdem aufgenommen. Soweit es nicht im Widerspruch zum dpANS steht, gibt es alles aus fig-Forth und FORTH-83 sowie einiges aus Laxen&Perry's F83. Die Fädelungstechnik ähnelt dem traditionellen "indirect threading" und die Datenstrukturen im Dictionary sind ähnlich den schon vor 15 Jahren von fig-Forth benutzten.

Stabilität ist eine Eigenschaft, ohne die kein System das Vertrauen der Benutzer erwirbt, das sie veranlaßt darauf aufbauend Stunden und Tage in ihre Entwicklungen zu investieren. Stabil ist aber nur ein einfaches System. Ein weiterer Grund, bei den Designentscheidungen auf bewährte und verstandene Konzepte zu bauen.

1.1 Was fehlt

Im Interesse der Einfachheit und Stabilität wurde auf verschiedene Dinge bewußt verzichtet, weil sie eine Größenordnung mehr Komplexität in das System hineingetragen hätten oder im Widerspruch zum Ziel der Portabilität stehen:

- die Fähigkeit sich selbst zu übersetzen
- ein Assembler
- die Fähigkeit, Maschinencode zu erzeugen
- die Fähigkeit, Libraries aufzurufen.[4]
- Multitasking

[4]Speziell für Linux existiert eine Lösung von Kevin Haddock.

2 Design

Schreibt man ein Forth-System in C, werden viele Dinge, die bei einer Assembler-Implementation Kopfzerbrechen bereiten können, sehr einfach. So ist ein in C programmierbares System von vornherein zur Fließkommaarithmetik, Freispeicherverwaltung usw. fähig. Sehr viele Forth-Worte brauchen also nur den entsprechenden Funktionsaufruf zu tun. Es gibt aber auch Bereiche, in denen C gegen den Strich zu bürsten ist, damit Forth herauskommt.

Ich möchte mich hier auf die zu bewältigenden Schwierigkeiten konzentrieren. Damit hoffe ich das Verständnis der entsprechenden Passagen im Quelltext zu erleichtern. Es wird sich allerdings zeigen, daß bei Beachtung einiger einfacher Regeln Änderungen und Erweiterungen am C-Quelltext leicht möglich sind, auch ohne sich in die Tiefen der Exception-Behandlung einzulesen.

2.1 Datentypen

Im Gegensatz zu Forth ist modernes C eine streng typisierte Sprache[5]. Forth macht zwar keinen Unterschied zwischen Adresse und ganzer Zahl, muß aber diese Laxheit ausgleichen durch präzisere Forderungen an das Verhältnis der wenigen elementaren Typen zueinander.

Implementiert man dpANS-Forth in ANSI-C, muß man es beiden recht machen. Im C-Quelltext folgen daraus viele explizite Typumwandlungen ("type casts"). Diese ansonsten bedenkliche Praxis führt hier nicht zu Problemen, weil die konsistenten Anforderungen von Forth an die beteiligten Datentypen eingehalten werden.

Insbesondere fordert der dpANS, daß eine Adresse und eine einfach genaue ganze Zahl dieselbe Größe im Speicher haben, und nennt solch eine Einheit *"cell"*. C erlaubt uns darauf einzugehen, indem wir unter mehreren ganzzahligen Datentypen denjenigen auswählen, der dieselbe Größe wie ein Zeiger hat. Dieser Datentyp wird zum ganzzahligen Grundtyp unter dem Namen `Cell`.

[5]Wer's nicht glaubt, versuche einmal ein größeres Programm *ohne Warnungen* durch mehrere verschiedene ANSI-C-Compiler zu bringen.

Der Standard fordert neben dem ganzzahligen Grundtyp noch einen doppelt so großen ganzzahligen Datentyp.[6] Nachdem wir aber für Forth' ganzzahligen Grundtyp schon *irgendeinen* ganzzahligen C-Typ nehmen mußten, können wir nicht davon ausgehen, daß der C-Compiler unter dem Namen `long` noch etwas größeres für uns hat. Die Arithmetik mit Forth' doppelt genauen Zahlen muß also unter Verwendung von `Cell` allein nachgebildet werden. Bei der Gelegenheit fällt es dann auch nicht weiter schwer, die in Forth übliche Reihenfolge der beiden Zellen einer doppelt genauen Zahl auf dem Stack einzuhalten.

Bei den laufenden Systemen sind durchweg die einfach genauen Zahlen 32 Bit groß und die doppelt genauen Zahlen 64 Bit. Es spricht aber nichts dagegen, auf einem entsprechenden System 64 Bit große einfach genaue und 128 Bit große doppelt genaue Zahlen zu haben. Eine Reduktion des Systems auf 16/32 Bit wäre – aus anderen Gründen als der Arithmetik – schwieriger.

2.2 Die Forth-Maschine in C

Es sollen mögliche Implementationsvarianten abgewogen werden, die gewählte erklärt und begründet werden. Dazu wird das Modell einer virtuellen Maschine herangezogen, das vom Forth-79 Standard definiert[7] aber auch schon in fig-Forth verwendet wurde.[8]

Der dpANS erwähnt die virtuelle Maschine nicht mehr. Er stellt sich auf einen abstrakteren Standpunkt und redet über die verschiedenen „Semantiken“ von Worten zur Übersetzungs- und Laufzeit. Das ist gut so, denn längst ist es gängige Implementationspraxis, die genaue Arbeitsweise der virtuellen Maschine zugunsten von prozessorspezifischen Optimierungen und persönlichen Vorlieben zu ignorieren, dabei nur mehr oder weniger genau darauf achtend, daß die oberste Ebene – eben die Semantik – noch genauso aussieht, wie man es von einem Forth erwartet.

Andererseits steht die virtuelle Maschine nicht im Widerspruch zum Standard und ist nach wie vor geeignet, ein Standard-System zu treiben.

Low-Level/High-Level Ein Forth-System muß elementare Grundoperationen bereitstellen. Normalerweise sind dies in Assembler geschriebene Worte, auch genannt Low-Level- oder Code-Worte oder Primitives. Geeignete Primitives

- manipulieren Daten- und Returnstack
- machen die Arithmetik
- führen bedingte und unbedingte Sprünge aus
- verarbeiten zur Laufzeit literale Konstanten der verschiedenen Typen
- greifen auf lokale Variablen zu

High-Level-Worte sind dagegen die compilierten Worte, die in irgendeiner Form aus Adress- oder Token-Listen bestehen, die vom *inneren Interpreter*, der Operation *next* der virtuellen Maschine, abgearbeitet werden.

Daneben gibt es noch die sogenannten “runtimes”. In Assembler-Implementationen sind das Code-Stücke, die in *next* enden, aber nicht als Primitives auftreten. Vielmehr bestimmen sie das Laufzeitverhalten von Variablen, Konstanten und vor allem von High-Level-Worten.

Nachdem der innere Interpreter (*next*) jedes Wort bedingungslos anspringt, muß für jedes Wort ausführbarer Code existieren. Die runtimes stellen diesen ausführbaren Code z.B. für Variablen bereit. Das runtime *nest* ist das ausführbare Stückchen Code für eine High-Level-Definition.

2.2.1 Die virtuelle Maschine

Ein klassisches “indirect threaded” Forth birgt in sich – ohne daß der Benutzer darauf direkten Zugriff hätte – eine virtuelle Maschine mit vier Registern, davon zwei Stackpointer in die beiden Stacks, und einigen Grundoperationen.

[6] Tatsächlich hat der Standard ein Herz und hat diese Forderung in einen optionalen “double number extension word set” ausgelagert.

[7] vgl. [1] S. 193

[8] eine detaillierte Diskussion dieser virtuellen Maschine und ihrer Grundoperationen realisiert auf verschiedenen CPUs findet sich in [4] S. 27

Die Register der virtuellen Maschine sind: Daten-Stack-Pointer *SP* und Return-Stack-Pointer *RP*, der Instruction-Pointer *IP* und ein weniger einleuchtendes aber wichtiges Register namens *W*, das stets in das Datenfeld[9] der aktuell ausgeführten Code-Definition zeigt.

Um all das zu erreichen, was der dpANS vorschreibt, genügt es nach wie vor, wenn ein Forth-System – neben einer geeignet ausgewählten Menge von Primitives – die elementaren Grundoperationen der virtuellen Maschine realisiert:

next eine gewisse Menge (je weniger, desto besser) von Maschinenbefehlen, die zum nächsten auszuführenden Primitive springen. Wohlgemerkt ist das im klassischen Forth *kein* Unterprogrammsprung, sondern ein Weiterspringen, von dem es keine Rückkehr gibt. Die Kontrolle über den Programmfluß wird durch die anderen Grundoperationen ausgeübt. Mit den Registern der virtuellen Maschine und einem Hilfsregister *X* läßt sich die Aktion *next* so beschreiben:

$$\begin{aligned} W &\leftarrow (IP) \\ IP &\leftarrow IP + 1 \text{ Cells} \\ X &\leftarrow (W) \\ W &\leftarrow W + 1 \text{ Cells} \\ &\text{Sprung nach } X \end{aligned}$$

nest ist der zunächst von einer :-Definition ausgeführte Code. Legt den Instruction-Pointer *IP* auf dem Return-Stack ab und setzt den *IP* auf den Beginn des Datenfeldes der aktuell ausgeführten Definition:

$$\begin{aligned} RP &\leftarrow RP - 1 \text{ Cells} \\ (RP) &\leftarrow IP \\ IP &\leftarrow W \\ &\text{Aktion von } next \end{aligned}$$

Next führt nun das erste Wort aus der Liste von Worten aus, die das compilierte High-Level-Wort ausmachen.

unnest Umkehrfunktion von *nest*, kehrt zum rufenden Wort zurück indem der von *nest* gesicherte Wert vom Return-Stack zurückgeholt wird:

$$\begin{aligned} IP &\leftarrow (RP) \\ RP &\leftarrow RP + 1 \text{ Cells} \\ &\text{Aktion von } next \end{aligned}$$

does Mit *nest* verwandt ist eine Funktion der virtuellen Maschine zur Unterstützung von `CREATE...DOES>`-Definitionen. Ein mit einem Definitionswort DEF definiertes Wort W muß im Sinne einer high-level Definition compilierten Code ausführen, der aber nicht im Datenfeld des Wortes W steht, sondern irgendwo anders, vorzugsweise innerhalb des Wortes DEF.

Nur auf die Operation *unnest* hat der Programmierer direkten Zugriff, sie wird durch das Wort `EXIT` repräsentiert. Die übrigen Grundoperationen der virtuellen Maschine sind dem Forth-Programmierer nicht zugänglich. *nest* wird vom Compiler bei Bedarf eingeplant und *next* bildet das Ende jeder Code-Definition, tritt für den Programmierer nur dann in Erscheinung, wenn er selbst Code-Definitionen hinzufügt, also den in viele Systeme integrierten Assembler benutzt.

2.2.2 Erste Ideen zur Umsetzung in C

Wir werden einen C-Compiler nicht dazu bewegen können, unsere in C geschriebenen Primitives mit Befehlen entsprechend der *next*-Operation abzuschließen, wie man das in einer Assembler-Implementation macht. Außerdem können wir ihn nicht dazu überreden, irgendwo einfach hinzuspringen, ohne sich um die Rückkehr zu kümmern.

Ruft er ein als C-Funktion definiertes Primitive auf, wird er sicherlich auf einem Unterprogramm-Sprung bestehen. Dem Primitive wird er entsprechend einen Unterprogramm-Rücksprung anfügen. Der innere Interpreter wird in erster Näherung[10] zu einer Schleife:

```
for (;;) (*ip++) ();
```

[9] früher auch PFA genannt

[10] Die *W*-Systemvariable wird hier ebenso ignoriert wie mögliche Lösungen für die Operationen *nest/unnest*

IP ist hier ein Zeiger auf Zeiger auf Funktionen, die Primitives realisieren. Compilierte High-Level-Definitionen sind also Listen von Zeigern auf die Funktionen, die die hineincompilierten Primitives realisieren.

Die Operation `(*ip++)()` kann man sich als *next* vorstellen, die `for`-Schleife drumherum hebt den Effekt des Unterprogramm-Sprungs wieder auf.

Alternativen Will man die Schleife vermeiden, darf man das einzelne Primitive nicht zu einer C-Funktion machen. Daraus folgt, daß alle Primitives Teil einer einzigen C-Funktion sein müssen. Dies führt zur "giant switch"-Lösung: Hierbei werden der innere Interpreter und die Realisierung aller Worte in eine einzige C-Funktion geschrieben. Diese C-Funktion besteht im wesentlichen aus einem *switch*-Statement. Compilierte Forth-Worte sind Listen von Codes, deren jeder einen Zweig in dem *switch*-statement auswählt. Der innere Interpreter sieht unvollständig ungefähr so aus:

```
for (;;)
  switch (*ip++)
    {
      case DROP: sp++; break;
      case DUP:  --sp; sp [0] = sp [1];
                 break;
      ...
```

Mit etwas Glück ist diese Methode sehr schnell, nämlich wenn der Compiler die wesentlichsten Systemvariablen `sp`, `rp` und `ip`, die lokal zu der großen Funktion deklariert sind, in Prozessorregister legt.

Nachdem dpANS-Forth aber sehr umfangreich ist, wird diese Funktion entsprechend *sehr* groß. Es ist unwahrscheinlich, daß eine so große Funktion noch gut optimiert – oder überhaupt von allen Compilern akzeptiert – wird. Ausgliederung von aufwendigeren Worten in eigene Funktionen ist erschwert durch den Umstand, daß wichtige Systemvariablen lokal zur Haupt-Funktion sind. Jegliche Änderung erfordert das neue Übersetzen praktisch des gesamten Systems. Gründe genug, diese Variante nicht weiter in Betracht zu ziehen.

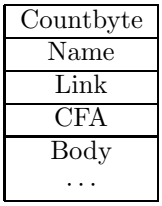
2.2.3 Probleme

Die erste Idee `for(;;) (*ip++)()` eignet sich bei näherer Betrachtung nur zur Ausführung von Worten, die ausschließlich aus Primitives zusammencompiliert sind. Denn es wird angenommen, daß compilierter Forth-Code aus Listen von Zeigern auf ausführbare Funktionen besteht. Ein High-Level-Wort ist aber keine ausführbare C-Funktion.

Die Lösung besteht natürlich in der Einführung einer weiteren Indirektion: Ein compiliertes Forth-Wort ist ein Zeiger auf einen Platz, in dem ein Zeiger steht, der über die Bedeutung des Wortes entscheidet. Der innere Interpreter wird zu:

```
for (;;) (**ip++) ();
```

Daraus und aus der angestrebten Traditionalität ergibt sich eine vorläufige Struktur für den Kopf einer Funktion im Dictionary:



Wie üblich werden also Name und Realisierung eines Wortes beieinander ins Dictionary geschrieben. CFA kann man jetzt als C-Funktions-Adresse lesen. Ihr Sinn ist derselbe wie in traditionellem indirect threaded Forth derjenige der Code Field Address: Dorthin wird verzweigt zur Ausführung des Wortes. Primitives haben hier die Adresse einer C-Funktion stehen, die die Semantik des Wortes bereitstellt und keinen Body. High-level-Definitionen haben hier einen Zeiger auf eine Runtime-Funktion (in C) stehen, die die im Body beginnende Liste von compilierten Adressen zu interpretieren beginnt: *nest*.

Wie kommt die Runtime-Funktion *nest* zu dem Body der Funktion, von der aus sie aufgerufen wurde? Sie kann ihn etwas trickreich aus *IP* schließen: `ip = &ip [-1][1]` (mit einigen Type-Casts). Besser man erleichtert ihr diese Aufgabe indem man im inneren Interpreter die Variable *W* der virtuellen Maschine einführt:

```
for (;;)
  {
    w = *ip++;
    (*w) ();
  }
```

Mit dieser endgültigen Form des inneren Interpreters lauten *nest* und *unnest*:

```
nest () {
  *--rp = ip;
  ip = ++w;
}
unnest () {
  ip = *rp++;
}
```

Ein weiteres Problem ergibt sich mit Definitionsworten. Wie gesagt ist für alle Definitionsworte der Code in der CFA eines von ihnen definierten Wortes derselbe. Das Datenfeld beginnt zwingend unmittelbar hinter der CFA. Wo also kann vermerkt werden, *von welchem* Definitionswort ein Wort definiert wurde?[11] Die wenig befriedigende Lösung besteht in der Einführung eines weiteren Zeigers `aux` in den Kopf jeder Definition, der im Fall einer von einem Definitionswort definierten Definition diese Information enthalten kann:

Countbyte
Name
Link
Aux
CFA
Body ...

Dieser Zeiger hilft glücklicherweise auch bei anderen Gelegenheiten: Er stellt z.B. den Zusammenhang her zwischen der compilierten Laufzeitsemantik von Kontrollstrukturen und der Definition, die diese Laufzeitsemantik compiliert hat. Dadurch kann der Decompiler `SEE` zu einem compilierten `?BRANCH` entscheiden, ob er von einem `IF` oder einem `WHILE` her stammt und eine entsprechend strukturierte Ausgabe erzeugen.

[11] Assembler-Implementationen generieren hier zur Laufzeit ein paar Maschinenbefehle, die in die CFA als ausführbarer Code eingetragen werden, und die in den high-level Code überleiten, der die Laufzeit-Semantik des Definitionswortes bereitstellt. Eine C-Implementation kann das nicht.

2.3 Das Dictionary

Das Dictionary ist in klassischen Forth-Implementationen der eine lineare Speicherbereich, in dem sich alle Forth-Worte – Namen und Realisierung – sowie statisch allozierte Daten in bunter Mischung beieinander befinden. Moderne Implementationen zeigen eine starke Tendenz, diesen Bereich aufzuteilen in Symboltabelle, einen Bereich für Maschinencode und einen für High-level-Definitionen und statisch allozierten Speicher.[12]

pfe – traditionell wie es ist – bleibt bei der einen großen Struktur für Namen und Daten. Lediglich der vom C-Compiler erzeugte Maschinencode liegt außerhalb.

2.3.1 Struktur

Die Namen werden sind wie üblich in einer linearen Liste organisiert. Zuviel der Semantik von Forth hängt von diesem Ansatz ab, als daß man ihn ohne größeres Kopfzerbrechen über Bord werfen könnte. Die Liste ist allerdings – weitgehend transparent – aufgeteilt wird in mehrere "Threads", wovon einer durch eine Hash-Funktion auf dem Namen des Wortes ausgewählt wird.[13] Man könnte andersherum sagen: Das Dictionary ist eine kleine Hash-Tabelle mit großem linear verkettetem Überlaufbereich. Es bedarf keiner komplizierteren Ansätze: Die so erreichte Geschwindigkeit der Dictionary-Suche ist voll befriedigend.

2.3.2 Wortlisten

Die Dictionary-Struktur kann vom Compiler nicht ohne großen manuellen Aufand etwa als initialisiertes Feld von `char` angelegt werden. Deshalb werden die Forth-Worte im C-Quelltext in einfacheren Tabellen organisiert, die die Zuordnung von Namen

[12] Die Motive dafür sind teilweise unseriös: Wenn es nämlich vor allem darum geht, einer ebenso misratenen wie verbreiteten Speicherarchitektur ein bißchen mehr Raum abzuringen.

[13] Diese Lösung habe ich in einem CSI-Forth gesehen.

und auszuführendem Code sowie Eigenschaften wie `IMMEDIATE` enthalten. Beim Systemstart wird aus diesen Tabellen das Dictionary gebaut.

Diese Tabellen, eine zunächst technisch motivierte Vorstufe des Dictionary, können auch als "Word Sets" aufgefaßt werden, wie sie der Standard kennt. Sie erlauben, den Quelltext des Systems modular aufzubauen entlang der Vorgabe des Standards, jeden Word Set in einer eigenen Eingabedatei, die abgeschlossen wird von einer Tabelle, die die in dieser Eingabedatei definierten Worte geschlossen bereitstellt und die Implementationsdetails kapselt.

3 Erweiterung

Einen Assembler bietet **pfe** nicht. Man kann es aber leicht durch C-Funktionen erweitern, was allerdings eine neue Übersetzung des Systems mit dem C-Compiler erfordert und insofern in der täglichen Arbeit ein schwacher Trost ist.

Dem C-Programmierer bietet der Rest des Systems folgendes Programmiermodell:

Datentypen: Mit den Objekten aus Sicht von Forth kompatible Objekte in C sind als die folgenden `typedefs` bekannt:

`Cell` eine Einheit auf dem Stack, Integer-Typ mit Vorzeichen, zuweisungskompatibel mit Zeigertypen.

`uCell` eine Einheit auf dem Stack, Integer-Typ ohne Vorzeichen.

`dCell` eine doppelt genaue Zahl mit Vorzeichen, `struct` bestehend aus `Cell hi` und `uCell lo`.

`udCell` wie `dCell`, vorzeichenlos.

`void (*pcode) (void)` ist ein Zeiger auf eine ein Primitive realisierende C-Funktion, die also der Gestalt `void primitive (void)` ist. Zur Vermeidung von Namenskollisionen enden die Namen all dieser C-Funktionen in einem Tiefstrich. Zur Erleichterung der Definition gibt es das Makro `code(NAME)`, das den Funktionskopf `void NAME_(void)` erzeugt. Im Kopf jedes Wortes steht ein solcher Zeiger auf die auszuführende Funktion.

`pcode *CFA` ist der Typ des Execution Tokens, ein Zeiger auf die Position im Kopf eines Wortes, wo der Zeiger auf dieas Wort realisierende C-Funktion steht.

Register der virtuellen Maschine sind global deklarierte C-Variablen:

`Cell *sp` Daten-Stackpointer

`CFA *ip` Instruktionszeiger. Zeigt in die Liste compilierter Worte, deren jedes jeweils genau das Execution Token, also ein Zeiger des Typs `CFA` ist.

`CFA **rp` Der Return-Stack dient zur Sicherung des `ip`, der Stapelzeiger ist also ein Zeiger auf den Typ des `ip`.

`CFA w` Das W-Register der virtuellen Maschine, hier kann ein Primitive seine eigene CFA ablesen. Zugriff auf den Body des Wortes über `(`*typ* `*)&w[1]`.

Systemvariablen Weil die Art der Speicherung der Systemvariablen sich ändern könnte, gibt es zum Zugriff auf die in Forth definierten Systemvariablen Makros mit nahenliegenden Namen. So kann in C-Funktionen z.B. auf den DP unter dem Namen `DP` zugegriffen werden usw.

Funktionen Das System enthält hunderte von C-Funktionen, wovon natürlich nicht jede in ihrer jetzigen Form garantiert – oder gar dokumentiert – werden kann. Beispiele:

`void abortq (char *fmt, ...);` gibt – mit den Fähigkeiten von printf() ausgestattet – eine Fehlermeldung aus und führt `-2 THROW` aus.

`void type (char *s, Cell n)` gibt eine Zeichenkette wie `TYPE` aus.

`char *word (char del)` wie das Forth-Wort `WORD`, liest ein Wort aus dem Eingabestrom in den Speicher ab `HERE` und übergibt diese Stelle als Funktionsergebnis.

`void run_forth (CFA xt);` führt ein beliebiges (auch High-level) Forth Wort aus. Enthält den inneren Interpreter.

Makros erleichtern die Anpassung von C an Forth sehr, weil sich darin viele der notwendigen Type-Casts verbergen lassen. Einige Beispiele:

`POP(T,P,X)` liest ein Objekt des Typs `T` über den Zeiger `P` in die Variable `X`, unabhängig von den tatsächlichen Typen der Operanden. Erhöht `P` entsprechend.

`PUSH(T,X,P)` Umkehrung zu `POP`, schreibt ein Objekt `X` des Typs `T` in einen zu kleineren Adressen hin wachsenden Stapel mit Stapelzeiger `P`.

`RPUSH(X)` legt ein Objekt der Größe einer Cell auf den Return-Stack.

`RPOP(X)` holt eine Cell vom Return-Stack.

`FLAG(X)` macht aus `X` ein Forth-Flag, also 0 bzw. -1.

`COMMA(X)` schreibt das Objekt `X` der Größe 1 Cell ins Dictionary.

Will man das System zum Beispiel um einen für ein bestimmtes Betriebssystem wichtigen Satz von Worten erweitern, bietet sich die Anlage einer entsprechenden Quelltextdatei an. Sie enthält alle system- oder aufgabenspezifischen C-Funktionen, die in eine Reihe von C-Funktionen münden, die die Gestalt von Forth-Primitives haben, also mit dem Makro `Code (`*name*`)` deklariert sind. Diese Primitives werden in eine Wortliste zusammengefaßt, die beim System registriert wird[14] und so beim Start von **pfe** mit ins Dictionary geladen wird.

Vollständiges Beispiel:

```
#include "forth.h"
#include "support.h"

Code (user_added_primitive)
{
  outs ("\nThis is a sample primitive."
        " See src/yours.c "
        "for it's definition.\n");
}

LISTWORDS (your) =
{
  CO ("USER-ADDED-PRIMITIVE",
      user_added_primitive),
};
COUNTWORDS (your, "Your kernel extensions");
```

Das Makro `LISTWORDS` deklariert eine Wortlisten-Struktur, die zu Initialisieren ist mithilfe der Makros:

`CO(NM,PCODE)`	gewöhnliches Wort
`CI(NM,PCODE)`	immediate Wort
`CS(NM,SEM)`	Compiler Erweiterung
`SV(NM,VAR)`	Systemvariable
`SC(NM,VAR)`	Forth-Konstante, deren Wert aus einer Variablen stammt
`OC(NM,VAL)`	normale Konstante
`VO(NM,RUNTM)`	`WORDLIST`

Das Makro `COUNTWORDS` zählt schlicht die Anzahl Worte in der Liste und stellt sie zur Einbindung bereit.

4 Ausblick

pfe wird sich entsprechend seiner Bedeutung weiterentwickeln. Zur Zeit scheint Bedarf an einem solchen System zu bestehen. Allerdings setzen die eingangs begründeten Design-Prinzipien der Entwicklung Grenzen. Was sich nicht mit überschaubarem Aufwand und portabel machen läßt, wird nicht mit Priorität behandelt. Es bleiben als *denkbare* Entwicklungsrichtungen:

- Bereitstellung eines Metacompilers auf **pfe**. Nicht mit dem Ziel **pfe** selbst damit zu übersetzen, sondern um es als Entwicklungsplattform für Mikrocontroller-Programmierung einsetzen zu können.
- Einbindung in eine grafische Benutzeroberfläche, evtl. das X-Window-System.
- Bereitstellung einfacher Grafikbefehle

[14] Zur Zeit durch Eintrag in die Tabellen innerhalb der Funktion `preload_dictionary()` in der Datei `dictnry.c`

- Nuetzliche Eigenschaften, wie z.B. die Speicherung compilierter Module.
- Multitasking.
- Bessere Unterstützung von noch mehr Rechnern.

Literatur

[1] American National Standards Institute: *draft proposed* American National Standard for Information Systems – Programming Languages – Forth (X3J14 dpANS-6 – June 30, 1993)

[2] Forth Interest Group: *fig-FORTH Installation Manual, Glossary, Model, Editor*, Version 1.3, San Carlos, CA, 1980

[3] FORTH Standards Team: *FORTH-83 Standard*, Mountain View, CA, 1983

[4] Zech, Ronald: *Die Programmiersprache Forth*, München: Franzis, 1983